Pseudo-Random Streams for Distributed and Parallel Stochastic Simulations on GP-GPU

Jonathan Passerat-Palmbach, Claude Mazel, David R.C. Hill

UMR 6158 CNRS

Pseudo-Random Streams for Distributed and Parallel Stochastic Simulations on GP-GPU

Jonathan Passerat-Palmbach [1] [*] [†] [‡],
Claude Mazel [*] [†] [‡],
David R.C. Hill [*] [†] [‡]

Originally published in: Journal of Simulation — June 2012 — pp 141–151
doi:10.1057/jos.2012.8

Abstract: Random number generation is a key element of stochastic simulations. It has been widely studied for sequential applications purposes, enabling us to reliably use pseudo-random numbers in this case. Unfortunately, we cannot be so enthusiastic when dealing with parallel stochastic simulations. Many applications still neglect random stream parallelization, leading to potentially biased results. In particular parallel execution platforms, such as Graphics Processing Units (GPUs), add their constraints to those of Pseudo-Random Number Generators (PRNGs) used in parallel. This results in a situation where potential biases can be combined with performance drops when parallelization of random streams has not been carried out rigorously. Here, we propose criteria guiding the design of good GPU-enabled PRNGs. We enhance our comments with a study of the techniques aiming to parallelize random streams correctly, in the context of GPU-enabled stochastic simulations.

Keywords: Stochastic Simulations; GP-GPU; Pseudo-Random Number Generators; Random Stream Parallelization

[1]This author's PhD is partially funded by the Auvergne Regional Council and the FEDER

[*] ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
[†] Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
[‡] CNRS, UMR 6158, LIMOS, F-63173 AUBIERE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

1 Introduction

Stochastic simulations can be computationally greedy applications and the tendency is to always tackle bigger problems following the increase of computer performances. Domain experts, trying to decrease over and over again their simulation execution time, look for new solutions. For about five years now, we have seen a growing interest in General-Purpose computation on Graphics Processing Units (GP-GPU) through the simulation community. GP-GPUs offer a startling amount of computational power at an incredibly low cost compared to computing clusters and supercomputers. However, the introduction of this new kind of architecture implies not only the rewriting of the simulation algorithms, but also of the tools they use. In this article, we have chosen to focus on Pseudo-Random Number Generators (PRNGs), which are the basis of any stochastic simulation.

Sequential PRNGs have been studied for a long time L'Ecuyer (2010), and finding a good quality PRNG to use in a sequential application has not been a problem for more than a decade. Many works have been accomplished to characterize the statistical quality of PRNGs, leading to several testing libraries. Nowadays, reference testing suites are well known and are used to assess PRNGs. However, a PRNG should always be considered in relationship with the scope of the application it feeds. For instance, cryptographic applications developers ought to base their choice on the NIST testing battery Rukhin et al. (2001), whereas simulationists should use TestU01 L'Ecuyer and Simard (2007). These two testing batteries can be considered as a standard for their respective domains at the time of writing.

According to Coddington (1996); Hellekalek (1998b), a PRNG should perform well on a single processor before being parallelized. Yet, statistical quality is a necessary but not sufficient condition when selecting a PRNG to use in a parallel context: indeed, parallel streams should be independent. Thus, providing high quality random numbers becomes even more difficult when dealing with parallel architectures. We have to take into account the parallelization technique: how will we partition random streams among parallel processing elements (threads or processors for instance)? How will we ensure the independence between parallel streams in order to prevent the simulations involved from producing biased results? The major problem concerning independence between random streams is that no mathematical proof exists to ensure it. However, some studies lay out well-known techniques to spread random streams through parallel applications Coddington (1996); Hellekalek (1998a); Traore and Hill (2001); Hill (2010); Reuillon et al. (2011). They try to ensure the maximum independence between random streams using different strategies. We will consider in this article whether or not, and how, these techniques can be implemented on GPU.

Apart from the parallelization technique, another point relies directly on the architecture where the involved stochastic simulations run. If we consider a GP-GPU environment, a new difficulty comes into play: harnessing the power of the device requires a rather good knowledge of GPUs. With recent programming frameworks like CUDA (Compute Unified Device Architecture) or OpenCL (Open Computing Language), almost anyone can develop applications for GPUs, but obtaining the announced performance gain implies a higher level of understanding. Most of the work and considerations exposed here rely principally on NVIDIA CUDA solutions. We are also working with the emerging OpenCL standard Khronos OpenCL Working Group (2010), but the latter is still not robust enough in our opinion. Its current performances are slower than what you could obtain with CUDA Karimi et al. (2010). However, we feel that this standard deserves our interest, and should take on an important part of our future work.

In this article, we will name as processing elements those effectively computing data in parallel. In a CUDA GP-GPU environment, threads will be regarded as these elements, since this framework relies on a thread level logic referred to as SIMT (Single Instruction, Multiple Threads). The latter abstracts a much more standard designation known as SIMD (Single Instruction, Multiple Data). GPUs are based on this kind of parallel architecture. Here, each thread must be given different data that will be computed by an identical operation. In the case of parallel stochastic simulations, we need to furnish an independent stochastic stream to each thread, in order to prevent potential biases that could be introduced otherwise. Now, the two subjects we are tackling in this study are:

- How are these random streams produced on GP-GPU?
- How can we ensure that they are independent?

These two main problems led us to share our experience of using stochasticity in GP-GPU-enabled simulations. We focus on the parallelization techniques of pseudo-random streams used to directly feed parallel simulation programs running on GPU (called kernels in the CUDA language). Before stating what we will present in this paper, let us point out first, that we will not propose any new PRNG, and second, that our study is not tied to the parallelization of random number generation algorithms, albeit we do survey some parallel PRNG algorithms. The main point of this study is to give guidelines that will hopefully help developers to use reliable parallelization techniques of random streams to ensure their independence, and that are, in addition, well adapted to GPU architectures particularities.

Some works have attempted to speed-up generation using the GPU before retrieving random numbers back onto the host. However, current CPU-running PRNGs display fairly good performances thanks to dedicated compiler optimizations. For instance, Mersenne Twister for Graphics Processors (MTGP) Saito (2010), the recent GPU implementation of the well-known Mersenne Twister Matsumoto and Nishimura (1998), is only announced as being 6 times faster than the CPU reference SFMT (SIMD-oriented Fast Mersenne Twister) Saito and Matsumoto (2008), which is already very efficient in terms of performance. In previous studies, we showed that the time spent in generating pseudo-random numbers consumes at most 30% of CPU time for some "stochastic-intensive" nuclear simulations Maigne et al. (2004), but they are very scarce. For less intensive simulations, when less than a billion numbers are needed, there is no real need for parallelization and unrolling is still the most efficient technique Hill (2003). Considering the small part of the execution time used by most stochastic simulations to generate random numbers, it is not worth limiting GPUs usage at the generation task. To use GP-GPUs at their full potential, we are more interested in providing random numbers to GPU-running applications that will consume them directly on the device.

In this study, we will:

- Propose GP-GPU specific criteria for PRNGs design;
- Survey random streams parallelization techniques of random streams in any distributed environment;
- Suggest requirements to implement parallelization techniques of random streams on GP-GPU;
- Study, according to the previously introduced requirements, the suitability of well-known PRNGs and random streams parallelization techniques for GP-GPU architectures.

2 Pseudorandom Number Generation on GPU

This section will survey the major propositions that can be found in the literature about PRNGs implementations on GPU platforms. Historically, a common way to deal with random numbers on GPU was to generate them on CPU before transferring them on the graphics processor. This solution had to face the well-known bottleneck of data transfer between the CPU host and the GPU device. Even with nowadays PCI Express 16X running at 8GB/s, this approach is considerably limiting the throughput of high performance applications.

Let us first focus on PRNGs implementations using GPUs as an hardware accelerator only, to provide random numbers quicker to host applications. Actually, such implementations can mostly be found during the genesis of GP-GPU, when the underlying architecture and programming languages features narrowed applications' scope. Until recently, designing a PRNG for GPU-enabled platforms could be very tricky as it forced programmers to deal with graphics Application Programming Interfaces (APIs). Some implementations are presented in Sussman et al. (2006). The authors especially list the limitations of these GPU dedicated PRNGs due to the past weaknesses of the hardware. Limited output per thread or untruthful operations were part of the restrictions that made these PRNGs feeble for High Performance Computing (HPC) applications.

Since 2008 and the recent advances from NVIDIA, new GPU software and hardware architectures offer the precision and speed needed by many HPC applications. Langdon presents a minimal implementation of the standard Park Miller PRNG Park and Miller (1988) on an NVIDIA 8800 GTX GPU in its paper from 2008 Langdon (2008). He announces a speed up of more than 40 compared to his Intel 2.40 GHz CPU. One year later, he increased again the speed of his application by four Langdon (2009) by using the NVIDIA CUDA technology NVIDIA (2010b) with a Tesla T10 GPU. Nevertheless, we do not advise the use of the old Park Miller generator that has many known flaws, though it was still employed until recently in some widespread networking simulation software Entacher and Hechenleitner (2003).

CUDA has been designed to allow developers to easily harness the computation power of GPUs. In his first implementation, Langdon had to deal with a complex and unadapted graphics API. With CUDA, developers can program GPUs without wasting their time making algorithms and their data fit into graphics dedicated data structures, such as pixels shaders. Furthermore, CUDA does not propose a new programming language but only some C extensions, making it easier to learn for C familiars. CUDA-enabled graphics boards fulfil the requirements noted in the conclusion of the previously cited Sussman et al. (2006), with for instance an implementation of the IEEE 754-2008 floating point numbers standard. The current generation of boards based upon the Fermi architecture is now proposing configurable L1 cache, ECC memory and a considerable increase of performance in double precision, while owning twice as much cores as the earlier mentioned T10 processor. At the time of writing, MTGP Saito (2010) is to our knowledge the sole parallel PRNG that has been specifically designed to run on GPU. The algorithm is intrinsically parallel, and targets the first goal of random number generation on GPU: speeding-up numbers output.

Recent GPU architectures, such as Fermi, opened new development perspectives. Having larger and faster memory areas available per thread, and being able to use object-oriented features, applications have become more and more ambitious, so have PRNGs implementations. GPUs are now considered as a fully capable platform, able to run entire applications by themselves. To do so, developers need PRNG implementations for GPU that allow their applications to directly consume the issued numbers. Such a tendency can be observed with the increasing

number of available libraries for CUDA. Pseudo-Random numbers generation follow the same tendency, and we have noticed several contributions in the last two years.

We noted two main proposals in this domain: CURAND and Thrust::random. They both aim to provide a straightforward interface to generate random numbers on GPU. Introduced in the latest version, at the time of writing, of the CUDA framework, CURAND NVIDIA (2010a) has been designed to generate random numbers in a straightforward way on CUDA-enabled GPUs. The main advantage of CURAND is that it is able to produce both quasi-random and pseudo-random sequences, either on GPU or on CPU. The quasi-RNG and pseudo-RNG implemented are respectively Sobol and XorShift Marsaglia (2003). The API of the library stays the same no matter which kind of RNG is selected and the platform on which the application is run on.

Thrust::random is part of a GPU-enabled general purpose library called Thrust Hoberock and Bell (2010). This open-source project intends to provide a GPU-enabled library equivalent to standard general-purpose C++ libraries, such as STL or Boost. Classes are split through several namespaces, of which Thrust::random is an example. The latter contains all classes and methods related to random numbers generation on GP-GPU. Thrust::random implements three PRNGs, each through a different C++ template class. We find a Linear Congruential Generator (LCG), a Linear Feedback Shift (LFS) Tausworthe (1965) and a Subtract With Borrow (SWB) Marsaglia et al. (1990); Marsaglia and Zaman (1991). Although the latter PRNG is mentioned as Subtract With Carry in Thrust::random documentation, Marsaglia's original proposition is known as SWB. In spite of the known flaws laid out by all these generators, the library offers simple ways to combine them into better quality randomness sources, like L'Ecuyer's Tausworthe combined generators L'Ecuyer (1996).

We recently proposed our own pseudorandom number generation toolkit for GPU named ShoveRand Passerat-Palmbach et al. (2011). It distinguishes from its counterparts by introducing a meta-model that enables the description of every PRNG characteristics. Moreover, this meta-model is implemented exclusively through C++ compile-time template mechanisms, thus introducing no overhead at runtime. ShoveRand offers a common API to users, whichever PRNG they select, and it also guides developers who would like to integrate a new generator to the framework. The latter performs compile-time analysis on the provided source code to ensure that only PRNG implementations which public interface matches our guidelines will compile successfully.

Although GPUs bring much more peak performances than CPUs, they must be carefully programmed to deliver the expected power, and most PRNGs have to be rethought from scratch to leverage GPUs characteristics. In fact, GPU architectures combine a manycore approach with SIMD vector cores. As vector processors do, GPU-enabled algorithms need to repeat the same operation on different data to correctly exploit the device. This is the main reason of the recent dedicated PRNGs proposals.

Other quality implementations can be found in Bradley et al. (2011). This study does not propose any new PRNG but details how a small set of reference quasi-random and pseudo-random number generators (Sobol, MRG32k3a and Mersenne Twister) have been successfully ported to GPU through CUDA. We draw your attention to the fact that these fine RNGs led to different GPU implementations, depending on their characteristics. For example, the small memory footprint of MRG32k3a allows an instance per thread whereas Mersenne Twister's large data structure conduct to a block of threads implementation level. We will detail the different

options offered when implementing a PRNG on GPU in a further section, but the work of Bradley et al. (2011) implicitly distinguishes two RNG groups: those which CPU code can directly be ported to GPU, with very few changes, and those who need to be redesigned to fit with GPU constraints.

Implementing PRNGs in a way to draw numbers directly on GPU led us to think about the best design of such pieces of software, considering both PRNG characteristics and GPU constraints.

3 Implementing PRNGs on GPUs

3.1 GP-GPU specific criteria for PRNGs design

As a result of the SIMD parallelism between threads and of their graphics processors legacy, GP-GPUs are not equivalent to a set of standard processors used in parallel. These particularities introduce some constraints that need to be satisfied if we do not want to see the overall simulation performance drop significantly. Thus, we will introduce in this section the requirements we find compulsory for a PRNG to run efficiently on a GP-GPU architecture.

The main goal targeted when using GP-GPUs is to obtain greater speed-ups. But this kind of device has not been primarily designed to support general computations and is more inclined to perform some arithmetic operations. Nowadays GPUs still display different performances with single and double precision floating point numbers, in accordance with the IEEE 754-2008 standard. For instance, the previous generation of NVIDIA supercomputing-dedicated GPU, the Tesla T10, was known to deliver ten times less computational power when dealing with double precision floating point numbers rather than with simple precision. Even if the current cutting-edge GPU, the NVIDIA Fermi T20, has considerably reduced the gap between these two precisions (a factor 2 still exists), it is wise to remain cautious before using double precision operations on GPU. Most of the time, single precision floats are sufficient enough to handle random values contained in $[0 ; 1[$ and should consequently be favoured. This proposition leads us to our first criterion: *single precision floating point numbers should be preferred throughout the GP-GPU random number generation algorithm.*

Another legacy of graphics processors is the heterogeneous memory organization. To complete what has previously been said on this subject, let us recall that several memory areas are reachable by threads running on a GPU. The capabilities of these memories, i.e. their capacity and response time, depend on two characteristics. First, the more threads can reach a memory area, the slower it is. Indeed, registers allocated to a single thread are the fastest memory this thread will be able to communicate with. Close to the same speed, we find shared memory, reachable by a relatively small amount of threads, all belonging to the same block, and so running on the same core of the GPU. On the other hand, every thread running on the GPU, regardless of which core they are located on, can access a wide memory area, commonly called global memory. This latter area is far slower than its previous counterparts (a few hundreds of GPU cycles are necessary for a basic global memory access). Second, read-only memories are faster since they can fully benefit from cache mechanisms, contrary to read-write memories. In the light of these memory constraints, it is obvious that GPU PRNGs should be designed with particular attention to sparing costly memory accesses. Commonly, static parameters will take place in read-only areas, whereas dynamic elements such as state vectors will be handled at the

thread or thread group level. In a more formal way, the following is another criterion of good design: *the algorithm should be designed in a way to avoid global memory accesses.*

Taking into account the particularities of GPUs and their architecture, we have proposed two new requirements for PRNGs to run efficiently on such devices. They are summed up hereafter:

1. Single precision floating point numbers should be preferred throughout the GP-GPU random number generation algorithm;
2. The algorithm should be designed in a way to avoid global memory accesses.

3.2 Location of PRNGs' internal data structures on GP-GPU

We focus here on the implementation level of PRNGs on the GPU, which describes the memory area where the algorithm's data is located. In Passerat-Palmbach et al. (2010), we identified three implementation levels, mapped on the CUDA thread hierarchy: threads, blocks of threads and grid of blocks. These strategies directly impact the PRNG implementation, so as the parallelization technique coupled with it. Let us introduce them to understand the technical prerequisites about the subjects we are tackling in this study.

Obviously, new PRNG algorithms have to take advantage of GPU intrinsic properties. For simplicity purposes, we will briefly introduce the major concepts that rule GPU architectures, that is to say: heterogeneous memory hierarchy and thread organization. Notions described in this paragraph are represented in Figure 1. On the one hand, the thread organization is meant to maximize the performances of the application. Threads are bundled into blocks of threads to be assigned to one of the Streaming Multiprocessor (SM) of the GPU. SMs can mostly be considered as GPU equivalents to CPU cores. The important point is that they have their own thread scheduler, which champions threads which data are available. Selected threads then run on Streaming Processors (SP): the processing units of SMs. The matching notion of heterogeneous memories comes into play at this point. Threads can access several memories displaying various capacities, which we will discuss more thoroughly in a further section. For now, we just need to keep in mind the hierarchy of memory areas: i.e. the classification of memories based on their response time and visibility from threads. From the fastest to the slowest, threads can access: registers, shared memory, local memory and global memory. This enumeration does not take into account any constant memory since they cannot be written from kernel programs. Thus, they would not be able to store the produced random numbers. When registers and local memory are dedicated to a single thread, shared memory is visible to all the threads within a common block (inside a SM), while every thread can reach global memory.

This memory organization highly affects the PRNG's performances. With the first considered implementation level, using a generator per thread, the internal state of the PRNG has to be saved in each thread's local memory. Most of the time, internal states are formed by several elements contained in arrays. Now, CUDA related works, like Kirk and Hwu (2010), specify that arrays declared within a thread are stored in the local memory. Although its name seems to indicate a thread scope, please note that local memory is actually a subset of global memory, and suffers consequently from the same slowness. This area is also allocated to threads when their register set is depleted, since registers are available in limited quantities on GPUs. At the time of writing, Fermi, the latest CUDA architecture has partially solved this problem, thanks to a larger amount of registers per thread, thus allowing small-memory-footprint PRNGs to store all their data in the register and shared memory space.

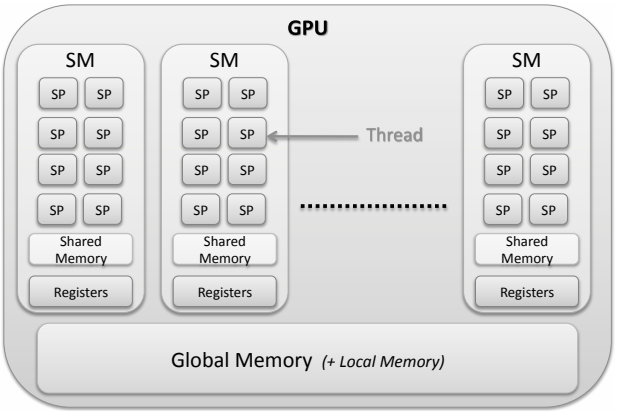

Figure 1: Simple representation of the major elements of a GPU

Equivalently, with the third cited implementation level, a PRNG for all threads, that is to say a grid-level scope, the global memory is solicited to store the state of the unique PRNG of the application. Each thread draws a number and updates its component of the state in global memory. In Zhmurov et al. (2010), authors present three basic generation algorithms working either with a single instance of the PRNG for the kernel or with an instance per thread. The three algorithms exposed are quite basic: Ran2, Hybrid Taus and a Lagged Fibonacci generator. In the same way, Langdon (2009) chooses to generate a number per thread in its GPU version of the Park-Miller algorithm. These two approaches make a heavy use of global memory. This has the advantage of being persistent across kernel launches within the same application. It is however important to realize that this area used to be quite slow: it implied a 400 to 800 clock cycle latency because it was not cached NVIDIA (2010b) in previous CUDA architectures. Once again, Fermi plays its part and enhances memory access time. However, the global memory approach will still display a worse latency than its counterparts.

The second implementation scope, the block of threads level, is the only one left to discuss. Every thread in a block can access a shared memory area. PRNG algorithms can consequently store their internal state in this space, enabling every thread of the block to update it. Shared memory is implemented on-chip and is consequently announced as fast as registers. Thus, PRNGs implemented at a block level will not suffer from the memory latency induced by slow global memory accesses. For independence purpose, we could imagine a variant of this block of threads scope, with a PRNG per warp.

The concept of Warps, as introduced by NVIDIA, corresponds to a subgroup of threads dynamically formed by the device at runtime: threads within a warp achieve memory accesses in parallel. Warps are thus the smallest GPU units that are able to process independent code sections. Indeed, given that different warps either run on different SMs, or on the same but at different clock ticks, they are fully independent to each other. Because of memory latency, warps-schedulers select the warps that have their data ready to process. Consequently, the more warps can be scheduled, the better the memory latency can be hidden.

In order to implement a PRNG at a given level, the following requirements must be met: first we need a common memory area accessible by every member of the group. In the case of warps, the shared memory area assigned to their belonging block will perfectly suit. Second, in order

to build their own random sequence, processing elements need to be able to distinguish their corresponding PRNG. There is no problem to do so when dealing with a PRNG implemented at either thread, block or kernel level, since CUDA provides us a way to uniquely identify each of the aforesaid element. Although warps identifiers are not directly available through CUDA keywords, we have shown in another study how a thread could fill a variable with the value of its parent warp's identifier [Passerat-Palmbach.etal.2011b].

The three main implementation scopes detailed in this section are sketched in Figure 2:

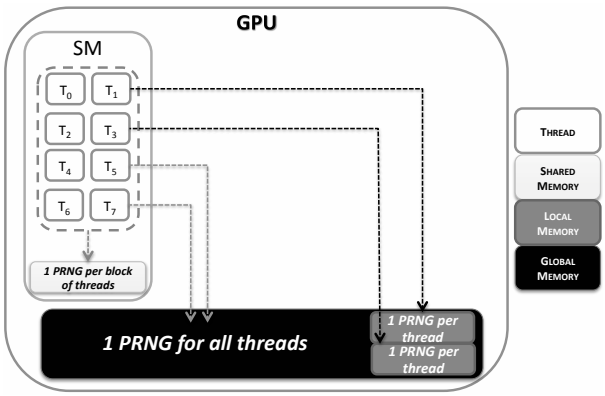

Figure 2: PRNGs implementation scopes and their location in the different memory areas of a GPU

As a conclusion, no matter which strategy we choose to implement random number generation facilities on GPU, we will always have to deal with distribution techniques. Such techniques could be hidden in a well-designed library, or directly applied by the simulation developer.

4 Requirements for distribution techniques of random streams on GP-GPU

4.1 GP-GPU specific requirements for random streams parallelization

The literature is full of references describing the profile of what a good usage of parallel PRNGs should be. For example, Coddington (1996); Hellekalek (1998b) list requirements that any sequential or parallel PRNG should meet. GPUs are particular parallel architectures, so any PRNG running on this kind of device should, at least, match the requirements enumerated in the previous references. In this section, we will successively check how these criteria can be adapted to GPU architectures.

Emphasizing parallel PRNG performances, Coddington (1996) noticed that *each processor should generate its sequence independently of the other processors.* We consider, indeed, that every processing element should have its own stochastic stream at its disposal. This condition must be satisfied first, not only for efficiency, but especially because GPU-enabled stochastic simulation parallelization principles rely on it. First, it is a necessary, but not sufficient, condition to fulfil to ensure a higher independence of stochastic streams feeding different replications of a simulation. Second, considering a single replication, the SIMT parallelism level leads threads to

compute their own data sets, including their own stochastic stream. Thus, our first requirement concerning random streams parallelization can be expressed as follows: *each thread should dispose of its own random sequence.*

As we explained previously, GP-GPU programming frameworks offer a thread scope rather than a processor one. The threads in use for GP-GPU propose an abstraction of the underlying architecture. They are concurrently running on the same device and handle their own local memory area. Thread scheduling is at the basis of GPU performances. Memory accesses are the well-known bottleneck of this kind of device. Indeed, running a large amount of threads in turn allows GPUs to bypass memory latency. There should always be runnable threads while others are waiting for their input data. Disregarding the effective number of processors, we theoretically say that the more threads you have, the better your application will leverage the device. Applications need to be written to use the maximum number of threads, but also to scale up transparently when the next GPU generation will be able to run twice as many threads as today. So, in accordance with Coddington (1996) who advocates that *the generator should work for any number of processors*, our second GP-GPU specific requirement for parallelization techniques of random streams is that *it must be usable for any number of GP-GPU threads.*

Returning to the original requirements, we then find Coddington (1996) states that *parallel random streams produced should be uncorrelated.* This criterion is related to both PRNG intrinsic properties and to the parallelization technique set up. We previously stated that a PRNG candidate to parallelization should first perform well on a single processor. Thus, we will not take its intrinsic qualities into account here. However, no matter the worth of the used PRNG, the parallelization techniques must be used carefully, as it will be shown in the next section. Please note that this requirement is neither affected by GP-GPU architectures nor by programming frameworks. As a result, we will just recall it without modifying its expression.

Coddington (1996) also noted that *the same sequence of random numbers should be produced for different numbers of processors, and for the special case of a single processor.* Here, we understand that the PRNG output on each processor should not depend on the number of processors used. This point is very important and must be treated carefully when choosing a parallelization technique. For example, in a distributed environment containing several processors, a scheduler can govern execution. Depending on the scheduler algorithm and on the global system charge, parallel executions of different parts of a simulation might not execute in the same order. In such a case, it is compulsory for the PRNG output to be independent from the order in which simulations parts may run. If this requirement is not met, reproducibility of simulations executions is no longer ensured. We can do this for games, but not for scientific applications. Reproducibility is needed when dealing with stochastic simulations, in order to debug a problematic case raised by a particular random stream for instance. We also think about design of experiments for simulations, where reproducibility is mandatory to isolate the impact of parameters variations on results.

In the case of GP-GPU, we find exactly the same problem at the thread level. These entities are also scheduled, not atomically but by bundles. Fortunately, both threads and their bundles own a unique identifier allowing us to distinguish them among executions. Thus, if a parallel random stream is only bound to the unique identifier of a thread, according to our first requirement, output will be reproducible through multiple executions. Therefore, we can obtain the necessary bijective relation between a T_i thread and an SS_i stochastic stream, as stated in Figure 3:

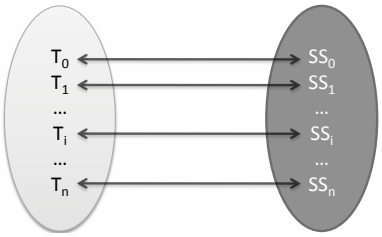

Figure 3: Bijective relation between threads and stochastic streams

Finally, we can write our last requirement in such a way: *when the status of the PRNG is not modified, the sequence of random numbers generated for a given thread must be the same no matter the number of threads and no matter of threads scheduling.*

Let us now sum up the requirements targeting GPUs we highlighted in this part, continuing the enumeration started in the previous section:

3. Each thread should dispose of its own random sequence;
4. The parallelization technique must be usable for any number of GP-GPU threads;
5. The parallel random streams produced should be uncorrelated;
6. When the status of the PRNG is not modified, the sequence of random; numbers generated for a given thread must be the same no matter the number of threads and no matter of threads scheduling.

5 Random streams parallelization techniques fitting GP-GPUs

This section presents how the main techniques used to distribute random streams between processing elements can be adapted to GPU architectures, depending on their ability to fulfil the previously introduced requirements.

5.0.1 Sequence Splitting

The Sequence Splitting (SS) method is also known as "Blocking" or "Regular Spacing". It consists in allocating non-overlapping, contiguous and equally sized blocks from the original random stream to form substreams. When partitioning a sequence $(x_i,\ i\ =\ 0,\ 1,\ \ldots)$ into N streams, the j^{th} stream is $x_{k+(j-1)m}, k = 0, \ldots, m-1$, where m is the length of each stream; m must be chosen so that each stream is long enough to achieve the stochastic simulation performed by the corresponding process. For instance in Hechenleitner (2004), Hechenleitner showed that in the OMNeT++ simulation software, the spacing between sequences set to 1 million draws led to biased results (due to inter-sequence correlations) for processes using more random numbers.

This technique implies the knowledge of how many numbers each thread will consume at most. Indeed, knowing that each thread consumes at most L random numbers, then the first L

numbers will be attributed to the first thread, the L following to the second thread and so forth. Following the previous formalism, we have $Y_i = \{X_{iL}, X_{iL+1}, \ldots, X_{(i+1)L-1}\}$. This numbers repartition is described in Figure 4.

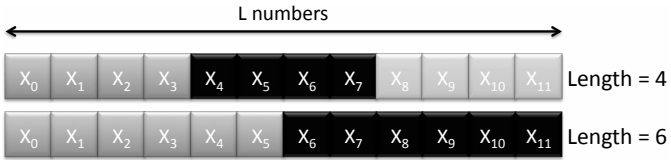

Figure 4: Two random streams parallelizations based upon Sequence Splitting with two different sub-sequences lengths

Efficient Sequence Splitting relies on a particular feature of the PRNG called Jump Ahead, or Skip Ahead. Here, we discern two categories of algorithms. Some PRNGs contain an algorithm able to perform Jump Ahead, thanks to intrinsic properties of the PRNG Haramoto et al. (2008). This allows us to reach any part of the sequence in equal time, regardless of the destination point. Let this technique be considered as Intrinsic Jump Ahead hereafter. The other solution is to emulate Skipping Ahead: to do so, we have to compute an advanced state by processing step by step the previous ones (for example, starting from $X_{iL}, X_{(i+1)L}$ can be computed by running the PRNG L times in a sequential way in order to get $X_{iL+1}, ..., X_{(i+1)L-1}$ and finally $X_{(i+1)L}$). In fact, whichever PRNG you use, you can unfold the sequence to the desired point, and store the state vector at this point in order to be able to load it later. Such a vector is named Seed Status in Passerat-Palmbach et al. (2010), since it is able to set a generator in a predefined state. Emulated Jump Ahead can become very costly though: indeed, the further you need to go in the sequence, the more time it takes to compute the Seed Status.

When an intrinsic Jump Ahead algorithm is available for the involved PRNG, Sequence Splitting is a very good approach for GP-GPUs. However, as far as we know there are few GP-GPU ports of algorithms with Jump Ahead features. At the time of writing, we are only aware of an MRG32k3a implementation detailed in Bradley et al. (2011) and of the recent Tiny Mersenne Twister (TinyMT) Saito (2011), available to download but not described in any scientific paper yet. On the other hand, Emulated Jump Ahead is not GPU-compliant because statuses computation is a purely sequential operation (we need X_n to compute X_{n+1}). As a consequence, different threads may lead to different computation time to process the next state to jump to, because of their sequential execution. Thus, the SIMD parallelism would be shrunk, making the overall gain decrease. To solve this problem, we propose to pre-compute substreams on the host side, store the Seed Status at each substream starting point and then transfer all these statuses to the device.

5.0.2 Random Spacing

The Random Spacing (RS) or Indexed Sequences (IS) method builds a partition of N streams by initializing the same generator with N random statuses. In the case of old LCGs it was named random seeding. For modern generators with a more complex status, the random statuses are generated with another RNG, and this technique is interesting when generators have a huge period. This technique is easy to set up. In Wu and Huang (2006), the authors have given the minimum distance between N sub-sequences which status were randomly chosen: it is in average equal to $1/N^2$ multiplied by the period length. More precisely, the probability of overlapping

between N sequences of length L, issued from a PRNG of period P is equal to: 1 – (1 – $NL/(P-1))^{(N-1)}$. That is equivalent to $N(N-1)L/P$ when $NL/(P-1)$ is in the neighbourhood of 0.

Overlapping risks become sizeable with short period PRNGs. All the PRNGs tested by Pierre L'Ecuyer and Richard Simard in L'Ecuyer and Simard (2007) display periods P from 2^{24} to 2^{131072}. Most of them have $\log_2 P$ of the order of a few dozens. Now, given that nowadays longest simulations can consume up to several hundreds of billions of random numbers, ($L = 10^{11}$), a hundred of such replications ($N = 100$) leads to a $N(N-1)L$ of approximately 10^{15}, i.e. 2^{50}. The probability to see an overlapping between two sub-sequences issued from a PRNG of period P far bigger than 2^{50} is negligible.

Random Spacing initialization process consists to draw a random number (from another generator), and to set it as the seed of the considered PRNG. Consequently, it fits GP-GPUs well, since this operation can be done in parallel without any constraint. In Figure 5, we have sketched the use of Random Spacing to issue a random stream assigned to each of the 3 threads represented. Yet, the risk of overlapping between sub-sequences must be evaluated according to the amount and the length of the sub-sequences and to the period of the PRNG used. If we select generators with large periods, such as WELLs and Mersenne Twister, this risk is negligible.

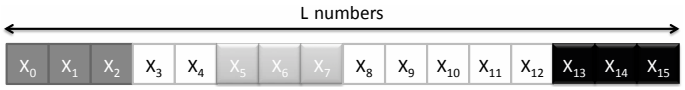

Figure 5: Random Spacing creation of three sub-sequences of equal length but differently spaced from each other

5.0.3 Leap Frog

Leap Frog (LF) is the way to partition a random stream like a deck of cards. Random numbers are allocated in turn to processors like cards would be dealt to players. Pragmatically, let each processor hold an i identifier. Every such PE will build a Y_i substream from an X original random stream such as $Y_i = \{X_i, X_{i+N}, \ldots, X_{i+kN}\}$, with N equal to the number of processors.

This technique is not quite adapted to split random streams on GP-GPU since it does not satisfy the last constraint expressed in the previous section. In fact, if the number of threads changes, the subsequence assigned to each thread will be different. This situation is shown in Figure 6. Now, the number of threads for an application is bound to the underlying device: GPUs can run a different number of threads concurrently, depending on their architecture generation. As a result, we would not be able to ensure the reproducibility of a simulation from a GPU to another, even if we initialized the PRNG with the same parameters. Furthermore, some bugs induced by random draws might not appear on the developer's device, while they would on the user's.

The solution would be to implement the PRNG at a level with a constant number of threads. The CUDA framework handles constant-sized bundles called warps, which always contain 32 threads at the time of writing. They are in fact a subdivision of blocks of threads, and access consequently the same shared memory area. Nowadays, a great number of GPUs do not have enough shared memory to store a PRNG status per warp. However, the newest GPU generations offer larger shared memory spaces that could potentially enable us to use Leap Frog following this idea.

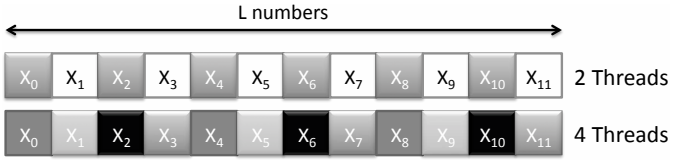

Figure 6: Different threads numbers leading to different random substreams through the Leap Frog method

5.0.4 Parameterization

Techniques presented so far tried to split a single stream into several substreams. Another approach consists in using several declinations of the same PRNG: each generator has the same structure and generation mechanism with a unique parameter set, called Parameterized Status hereafter.

Although no mathematical proof can establish this independence, some implementations of Parameterization are safe according to the current state of the art Mascagni and Srinivasan (2004). We especially think to the Dynamic Creator (DC) algorithm Matsumoto and Nishimura (2000) coming along with most of the generators from the Mersenne Twister family. DC integrates a unique identifier, which belongs to the Parameterized Status of the PRNG. This identifier becomes a part of the characteristic polynomial of the matrix that defines the recurrence of the PRNG. Two identifiers will consequently lead to two different Parameterized Statuses. Furthermore, DC ensures that the characteristic polynomials we obtain are mutually prime, and the authors assert that the random sequences generated with such distinct Parameterized Statuses will be highly independent, even if, as mentioned before, this fact cannot be mathematically proven.

This technique displays some constraints making it difficult to port to GPU. Unfortunately, few PRNGs propose it intrinsically. Some, such as LCGs, are even reckoned as bad candidates for Parameterization De Matteis and Pagnutti (1988). In addition, storing Seed Statuses (basically the common seed given by the user to initialize a generator, or the internal state vector of the PRNG) being already problematic, we can scarcely imagine spending vast amounts of memory to store a Parameterized Status per thread. A Parameterization example is proposed in Figure 7:

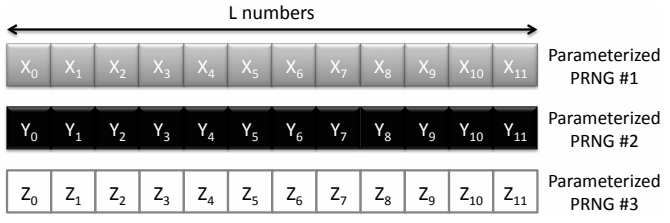

Figure 7: Three parameterized PRNGs producing three highly independent random sequences

Every technique introduced so far, presents advantages and drawbacks. Most of them are related to the chosen PRNG. Depending on the application and environment you own, you might be forced to select a PRNG knowing it has some flaws in particular cases. In this way, Table 1 states PRNG kinds and parallelization techniques that work well together:

Table 1: Summary of the potential PRNG/Parallelization technique associations

Technique	***Preferred PRNGs***	***PRNGs to avoid***
Leap Frog	None (disable reproducibility)	Linear generators
Sequence Splitting	Intrinsic Jump-Ahead compliant	Emulated Jump-Ahead
Random Spacing	Large period	Short period
Parameterization	MT family	LCG

6 Conclusion

We have seen that more and more applications, and especially stochastic simulations, tend to take advantage of recent GP-GPU architectures in order to improve their performance. However GPU computing needs to offer the same tools as other platforms. High quality PRNGs belong to this category and have existed for more than a decade, although some recent publications dealing with GP-GPU implementations of PRNGs still propose old and weak generators. In this paper we have shown how difficult it could be to obtain good quality pseudo random sequences on GP-GPU. Indeed, it implies taking into consideration two different domains: GP-GPU programming and PRNG parallelization techniques. Issuing a PRNG that can produce independent stochastic streams when used in parallel is a first hurdle that not all PRNGs can get over. When you have at your disposal one that fulfils this requirement, it has to be ported to GP-GPU. It means that you need to think about a GPU implementation of your PRNG, if it is not already available.

This paper introduced the main problems that you will be faced with when trying to port your stochastic simulations to GP-GPU. To avoid these difficulties, and above all, errors and performance drops that could result from the use of a hazardous GPU-enabled PRNG, we first proposed PRNGs criteria dedicated to GPUs. In order to take advantage of the existing PRNG parallelization techniques on GPU, we first defined a set of requirements that should be met by any PRNG implementing a distribution technique on GPU. Finally, we studied the applicability of widespread distribution techniques on GPU, and determined which techniques best matched GPU constraints. These requirements and chosen techniques sum up the experience accumulated in our research team concerning GPU-enabled stochastic simulations.

We encountered lots of parameters brought up by PRNGs and GPU programming. As long as it can dramatically impact both the overall performance of the simulation and the quality of its results, it might be a good point to propose a straightforward API to use well-defined PRNGs on GPUs. In this way, libraries laid out in this paper represent in our mind an elegant manner to do so. They allow every user to easily call PRNG functions without wasting time on how parameters should be set. In the same way, they enable advanced users to adjust parameters in their own fashion. As a matter of fact, we introduced a recent framework named ShoveRand, which embed PRNGs following the presently established requirements into a GPU-enabled library with a unified API. This library permanently evolves to integrate new PRNGs, such as the recent TinyMT from Mutsuo Saito, but also features proposed by the C++ Technical Report 1 to enhance random number generation.

References

Bradley, T., du Toit, J., Tong, R., Giles, M., and Woodhams, P. (2011). *GPU Computing Gems Emerald Edition*, chapter 16 - Parallelization Techniques for Random Numbers Generators, pages 231–246. Elsevier.

Coddington, P. (1996). Random number generators for parallel computers. Technical Report 2, NHSE.

De Matteis, A. and Pagnutti, S. (1988). Parallelization of random number gemerators and long-range correlations. *Numerische Mathematik*, 53:595–608.

Entacher, K. and Hechenleitner, B. (2003). Pitfalls when using parallel streams in omnet++ simulations. In *Interdomain Performance and Simualtion (IPS) Workshop*.

Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., and L'Ecuyer, P. (2008). Efficient jump ahead for f2-linear random number generators. *INFORMS Journal on Computing*.

Hechenleitner, B. (2004). *Defects in Random Number Routines of Well-Known Network Simulators and Appropriate Improvements*. PhD thesis.

Hellekalek, P. (1998a). Don't trust parallel monte carlo! In *Proceedings of Parallel and Distributed Simulation PADS98*, pages 82–89. IEEE.

Hellekalek, P. (1998b). Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5-6):485–505.

Hill, D. (2003). Urng: A portable optimization technique for software applications requiring pseudo-random numbers. *Simulation Modelling Practice and Theory*, 11(7-8):643–654.

Hill, D. (2010). Practical distribution of random streams for stochastic high performance computing. In *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, pages 1–8. invited paper.

Hoberock, J. and Bell, N. (2010). Thrust: A parallel template library. `https://github.com/thrust/thrust`. Version 1.3.0.

Karimi, K., Dickson, N., and Hamze, F. (2010). A performance comparison of cuda and opencl. `http://arxiv.org/abs/1005.2581v3`. submitted.

Khronos OpenCL Working Group (2010). The opencl specification. Specification 1.1, Khronos Group.

Kirk, D. and Hwu, W. (2010). *Programming Massively Parallel Processors*. Morgan Kaufmann.

Langdon, W. (2008). A fast high quality pseudo random number generator for graphics processing units. In *IEEE CEC 2008, Hong Kong*, pages 459–465.

Langdon, W. (2009). A fast high quality pseudo random number generator for nvidia cuda. In *GECCO'09*, volume 10, pages 2511–2514. ACM Press.

L'Ecuyer, P. (2010). *Pseudorandom Number Generators*. John Wiley & Sons, Ltd.

L'Ecuyer, P. (1996). Maximally equidistributed combined tausworthe generators. *Mathematics of computation*, 65(213):203–213.

L'Ecuyer, P. and Simard, R. (2007). Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–40.

Maigne, L., Hill, D., Calvat, P., Breton, V., Reuillon, R., Legre, Y., and Donnarieix, D. (2004). Parallelization of monte carlo simulations and submission to a grid environment. *Parallel processing letters*, 14(2):177–196.

Marsaglia, G. (2003). Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6.

Marsaglia, G., Narasimhan, B., and Zaman, A. (1990). A random number generator for pc's. *Computer Physics Communications*, 60(3):345–349.

Marsaglia, G. and Zaman, A. (1991). A new class of random number generators. *Annals of Applied Probability*, 3(3):462–480.

Mascagni, M. and Srinivasan, A. (2004). Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing*, 30(7):899–916.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 8(1):3–30.

Matsumoto, M. and Nishimura, T. (2000). Dynamic creation of pseudorandom number generators. In Niederreiter, H. and Spanier, J., editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer.

NVIDIA (2010a). *CUDA CURAND Library*. NVDIA Corporation.

NVIDIA (2010b). *NVIDIA CUDA Programming Guide Version 3.2*.

Park, S. and Miller, K. (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201.

Passerat-Palmbach, J., Mazel, C., Bachelet, B., and Hill, D. (2011). Shoverand: a model-driven framework to easily generate random numbers on gp-gpu. In *IEEE International Conference on High Performance Computing & Simulation*, pages 41–48.

Passerat-Palmbach, J., Mazel, C., Mahul, A., and Hill, D. (2010). Reliable initialization of gpu-enabled parallel stochastic simulations using mersenne twister for graphics processors. In *ESM 2010*, pages 187–195. ISBN: 978-90-77381-57-1.

Reuillon, R., Traore, M. K., Passerat-Palmbach, J., and Hill, D. R. (2011). Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with distme: Application to life science simulations. *Concurrency and Computation: Practice and Experience*. `http://dx.doi.org/10.1002/cpe.1883`.

Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. (2001). A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, NIST.

Saito, M. (2010). A variant of mersenne twister suitable for graphics processors. `http://arxiv.org/abs/1005.4973`. submitted.

Saito, M. (2011). Tiny mersenne twister (tinymt). `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html`. accessed 14 September 2011.

Saito, M. and Matsumoto, M. (2008). Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In Keller, A., Heinrich, S., and Niederreiter, H., editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, volume 2, pages 607–622. Springer Berlin Heidelberg.

Sussman, M., Crutchfield, W., and Papakipos, M. (2006). Pseudorandom number generation on the gpu. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 87–94. ACM.

Tausworthe, R. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19(90):201–209.

Traore, M. and Hill, D. (2001). The use of random number generation for stochastic distributed simulation: application to ecological modeling. In *Proceedings of 13th SCS-European Simulation Symposium*, pages 555–559.

Wu, P. and Huang, K. (2006). Parallel use of multiplicative congruential random number generators. *Computer Physics Communications*, 175(1):25–29.

Zhmurov, A., Rybnikov, K., Kholodov, Y., and Barsegov, V. (2010). Efficient pseudo-random number generators for biomolecular simulations on graphics processors. Technical report, CERN.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/